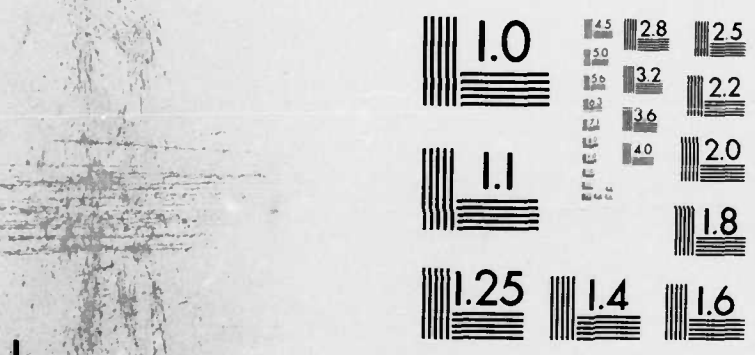
AD-A139 190

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO , CA
DESIGNING EFFICIENT COMMUNICATION NET-
WORKS BY: G CARLSSON UCSD

1 OF 1
NOSC CR 210
UNCLASSIFIED
OCT 1983



MICROCOPY RESOLUTION TEST CHART

(Based on NBS 1963A Target Design)

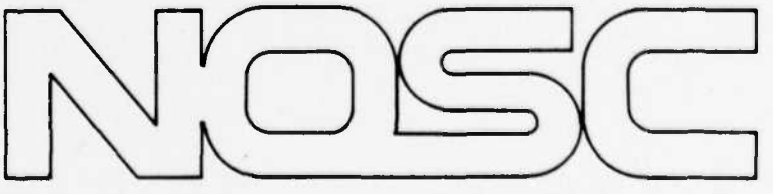
NOSC CR 210

Contractor Report 210

DESIGNING EFFICIENT COMMUNICATION NETWORKS

G. Carlsson
UCSD

October 1983
Final Report
June 1983 — August 1983



NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152

NAVAL OCEAN SYSTEMS CENTER SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

J.M. PATTON, CAPT, USN
Commander

R.M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

This task was performed for Naval Ocean Systems Center, Code 6322, San Diego, CA 92152, under program element 61152N, subproject ZR0000101 (NOSC 632-ZS96). This work was performed under contract N66001-83-C-0382 by the Department of Mathematics, University of California, San Diego, La Jolla, CA 92093. The contracting officer's technical representative was Harlan Sexton, NOSC Code 6322.

Released by
P.M. Reeves, Head
Electronics Division

Under authority of
R.H. Hearn, Head
Fleet Engineering Department

TJ

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NOSC CR 210	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DESIGNING EFFICIENT COMMUNICATION NETWORKS		5. TYPE OF REPORT & PERIOD COVERED Final October 1982–September 1983
		6. PERFORMING ORG. REPORT NUMBER 84-030
7. AUTHOR(s) Gunnar Carlsson		8. CONTRACT OR GRANT NUMBER(s) N66001-83-C-0382
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Mathematics University of California, San Diego LaJolla, CA 92093		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N-ZR0000101 (NOSC 632-ZS96)
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Ocean Systems Center (Code 6322) San Diego, CA 92152		12. REPORT DATE December 1983
		13. NUMBER OF PAGES 42
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer architectures
Parallel processing
Communication networks
Systolic arrays
VLSI design
Graph theory

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In recent years it has become clear that communication time between processors is one of the most severe limiting factors in designing high speed parallel computers. Further, for massively parallel machines unrestricted communication via such methods as data buses is impractical. Thus, it seems sensible to investigate the design of networks which allow efficient communication between processors subject to the restriction that each processing element may 'talk' to a fixed small number (≤ 10, say) of others. This report examines this problem in both a purely heuristic way, and also investigates a theoretical method of attack.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Table of Contents

I. INTRODUCTION: SUMMARY OF RESULTS

In recent years, it has become clear that communication time between processors is one of the most severe limiting factors in designing high speed parallel computers. This being the case, it becomes necessary to design networks of processors in which the communication time is small. A mathematical version of this design problem is the problem of designing graphs of given fixed degree and number of vertices with small diameter. (The report contains definitions of these terms.) In this report, we have attacked this problem from two directions. First, we have constructed a heuristic algorithm which finds graphs with small diameter, and implemented it on a computer. The program is written in the C-language. We have compared the results with previous best known results. Secondly, we have studied a collection of graphs which have compact and systematic descriptions, the so-called Cayley graphs. These graphs have the advantage that routing algorithms for them are easy to specify. We have given a crude comparison of their diameter with that of a theoretical bound, and studied a specific class of them, the "modified cube-connected cycles."

The results we have obtained may be summarized as follows.

(a) The heuristic algorithm we have constructed is an improvement over all previous algorithms of its kind. In particular, we have improved many of the best known values for dense graphs of given degree and diameter. A complete description is given in section II-B, where Fig. 1 shows all the improved values we have obtained.

(b) Our algorithm does not find the densest known graphs in cases where they are constructed using systematic combinatorial constructions.

This suggests that one should study a restricted class of graphs, having systematic descriptions.

(c) The symmetric groups S_n admit graph structures whose diameters approach the theoretical bound (Moore bound) arbitrarily well as $n \to \infty$. This suggests that they should be studied much more carefully as a possible source of efficient communications networks.

(d) A slight modification of the cube-connected cycles of [P-V] produces an infinite family of graphs whose diameter grows as $2 \log_2(K)$, where K is the number of points in the graph. This compares favorably with $5/2 \log_2(K)$, which is the diameter of the cube-connected cycles.

(e) A layout is given for these modified cube connected cycles, whose area is 3/2 times the area of the cube connected cycles with the same number of nodes. The VLSI measure of complexity, AT^2, is thus in fact slightly decreased, by a factor of 24/25.

The substantiation for these results is obtained by running the heuristic program which we have designed, and by theoretical analysis contained in section II.

II. DETAILED DESCRIPTION OF RESULTS

(A) Measures of Communication Time

Throughout, we will be interested in arrays of "processors," connected by "wires." The nature of these processors is not specified, because we want to study the general problem of communication time, without restricting to a specific situation. We will formalize this notion by considering graphs Γ, where the vertices of the graph correspond to processors and edges to wires. We will suppose that the array functions is such a way that at every time information is allowed to flow along one wire. The time required to move along one wire is presumed to be constant. By the distance between two processors, or the corresponding vertices x and y, we mean the length of the shortest path in Γ from x to y. A path of length n from x to y is a sequence of vertices $\phi = \{v_0, \ldots, v_n\}$ in Γ, so that $v_0 = x$, $v_n = y$, and so that for each i, $v_i v_{i+1}$ is an edge of Γ. Clearly, this is the same as the length of the shortest sequence of wires connecting the processors corresponding to x and y. We denote the distance by $d(x,y)$. By the diameter of the array, we mean

$$\max_{x,y\in\Gamma} d(x,y) = \Delta(x,y) .$$

where the max is taken over all pairs of vertices in Γ. (From this point on, we no longer speak of the arrays of processors but only of their corresponding graphs.) We wish to study the problem of designing graphs with a given number of vertices, having small diameter. Of course, with no constraints on the graph, this is a trivial problem, since complete graphs all have diameter 1. However, technology dictates that the number

of edges from each vertex should be less than or equal to some finite number d. Accordingly, we say that a graph is regular of degree d if every vertex has precisely d edges coming from it, and we attempt to solve the problem of minimizing the diameter of regular graphs of degree d having, say, N vertices. There is an obvious a priori upper bound to N, given d, called the Moore bound (see [Bo]), which is

$$N \leq 1 + d + d(d-1) + \ldots + d(d-1)^{k-1}$$

where k is the diameter of the graph. This says that asymptotically, k grows at least as fast as

$$\log_{d-1}(N) = \frac{\ell n\ N}{\ell n(d-1)} .$$

It is known, however, that this bound is only sharp in a finite number of cases for $d \geq 3$ (for $d = 2$, the cyclic graphs are all examples where it is sharp), and it seems to be the case that it in fact is rather crude. Our efforts toward studying this problem consist of the construction of a heuristic algorithm (see §B), and some specific constructions derived from group theory (see §D).

The diameter is itself only a weak measure of the effectiveness of the processor array. Suppose, for instance, that the processors have no memory, and that one wishes to simultaneously transfer information from the vertex v_i to the vertex w_i, for $i = 1, \ldots, k$. Since there is no memory, one must produce a collection of paths $\phi^{(i)}$ of length n in Γ, with $v_m^{(i)} \neq v_m^{(j)}$ for all $1 \leq i, j \leq k$, and so that $v_0^{(i)} = v_i$, $v_n^{(i)} = w_i$.

Here, $\phi^{(i)} = \{ v_0^{(i)}, \ldots, v_n^{(i)} \}$. Such a collection of paths is called a k-separated multipath in Γ from $(v_1, \ldots, v_k)$ to $(w_1, \ldots, w_k)$. We define the k-separated distance between $(v_1, \ldots, v_k)$ and $(w_1, \ldots, w_k)$ to be the minimum length of all k-separated multipaths from $(v_1, \ldots, v_k)$ to $(w_1, \ldots, w_k)$ in Γ, and denote it by $d_k((v_1, \ldots, v_k), (w_1, \ldots, w_k))$. The k-separated diameter is

$$\max d_k \left((v_1, \ldots, v_k), (w_1, \ldots, w_k) \right) = \Delta_k(\Gamma)$$

where the max is taken over all pairs of k-triples, with $v_i \neq v_j$, $w_i \neq w_j$ unless $i = j$.

The k-separated diameter $\Delta_k(\Gamma)$ takes congestion into account, and so is a more sensitive measure of effectiveness of the processor array. However, it assumes that the processors have no memory. We will define another measure of efficiency, $\Delta_k(\Gamma,\ell)$, which assumes that each processor has ℓ units of memory. By a (k,ℓ)-separated multipath of length n from $(v_1, \ldots, v_k)$ to $(w_1, \ldots, w_k)$, we mean a k-tuple $\phi^{(i)}$ of paths of length n, so that

(a) $\phi^{(i)}$ is a path from v_i to w_i.

and

(b) Each of the k-tuples $\{ v_j^{(1)}, v_j^{(2)}, \ldots, v_j^{(k)} \}$ contains each vertex at most ℓ times.

Note that $\Delta_k(\Gamma,1) = \Delta_k(\Gamma)$, and that for $\ell \geq k$, $\Delta_k(\Gamma,\ell) = \Delta(\Gamma)$. Intuitively, $\Delta_k(\Gamma,\ell)$ measures the time required to transfer the information in any k-tuple of processors to any other k-triple of processors, given that each processor has ℓ units of memory.

We will now observe that for any graph Γ, $\Delta_k(\Gamma,\ell)$ is just the diameter of a graph associated with Γ. For, form the k-fold product graph $\Gamma x \ldots x \Gamma$. In $\Gamma x \ldots x \Gamma$, consider the full subgraph Γ_k^ℓ on the set $V_k^\ell \subseteq V x \ldots x V$, (here V denotes the vertex set of Γ), where $V_k^\ell = \left\{ (v_1, \ldots, v_k) \;\middle|\; \text{no vertex appears more than } \ell \text{ times} \right\}$. Then it is easy to see that $\Delta_k(\Gamma,\ell) = \Delta(\Gamma_k^\ell)$. Thus, we have reduced these more sophisticated measures of effectiveness to a diameter question; this will be a useful reduction in view of the algorithm to be considered in the next section.

The diameter is unfortunately often rather expensive to compute. Consequently, one would like to obtain some less sensitive, more easily computable invariants of graphs which still have some relation to the diameter.

Definition The girth of a graph Γ is the length of the shortest cycle of Γ.

We note that if a graph has diameter k, then its girth is at most $2k+1$. Intuitively, the girth is inversely related to the diameter. For a fixed number of points, small diameter tends to imply large girth. We summarize the facts known about girth relating to this problem.

(a) There is a lower bound for the number of points in a graph with a given girth, analogous to the Moore bound (see [B]).

(b) For a graph of ~~old~~ odd girth $g = 2k + 1$, and degree d, the number of points is at least

$$1 + d + d(d-1) + \ldots + d(d-1)^{k-1}$$

This bound is attained only for $d = 2$, or for $d = 3, 7$, and possibly $5, 7$, $g = 5$.

(c) For a graph of even girth $g = 2k$ and degree d, the lower bound is

$$1 + d + d(d-1) + \ldots + d(d-1)^{k-2} + (d-1)^{k-1}$$

This bound is known to be attainable only for $g = 4$, 6, 8, or 12. Also, only $d = p^s + 1$ are known to occur, where p is a prime. The diameter is in each case k. These graphs provide by far the optimal graphs for the diameter problem with given diameter and degree.

We define certain numbers related to the girth. Given a vertex $x \in \Gamma$, we define $N_k(x)$ to be the number of vertices connected to x by a path of length k, and define $\nu_k(\Gamma)$ to be $\min_{x \in \Gamma} N_k(x)$. Note that if Γ is a regular graph of degree d, then $\nu_k(\Gamma)$ is bounded above by $1 + d + d(d-1) + \ldots + d(d-1)^{k-1} = \mu_k(d)$, and that the girth of Γ is $\geq \ell$ if $\nu_k(\Gamma) = \mu_k(d)$ for all $k \leq \ell/2$. Consequently, to maximize girth, one should attempt to maximize successively $\nu_2(\Gamma)$, $\nu_3(\Gamma)$, ..., $\nu_k(\Gamma)$, $\nu_{k+1}(\Gamma)$, ... In each case, $\nu_{k+1}(\Gamma)$ should be maximized subject to the constraint that one remains at an optimum for the previous values of the subscript. ν_k is quickly computed for small values of k, and the ν_k's are another more computable collection of invariants of graphs, which are related to the efficiency of the associated array of processors. This is particularly useful in attempting to work with the measures $\Delta_k(\Gamma,\ell)$, since the graphs Γ_k^ℓ are usually quite large, so the time spent computing invariants is of primary importance.

(B) A Heuristic Algorithm

In this section, we produce a "hill-climbing" algorithm using a particular heuristic criterion to find graphs of a given fixed degree and find fixed number of points with small diameter.

Let Γ be a graph of degree d, and let v_1, v_2, w_1, w_2 be vertices of Γ so that v_1v_2 and w_1w_2 are edges of Γ. Then by the perturbed graph based on $\{v_1, v_2, w_1, w_2\}$, we mean the graph $\tilde{\Gamma}$ whose vertices are the same as those of Γ, and whose edge set is $\left(E_\Gamma - \{(v_1,v_2), (w_1,w_2)\}\right) \cup \{(v_1,w_1), (v_2,w_2)\}$. Here E_Γ is the edge set of Γ. We say also that $\tilde{\Gamma}$ is the result of a perturbation on Γ. These modifications are precisely the X-changes defined in [T-S]. $\tilde{\Gamma}$ is regular of degree d if Γ is. We will view these perturbations as "small" change in the graph, and move in directions which improve a certain functional which we will define below.

Graphs will be encoded by their "incidence matrices." We number the vertices of the graph Γ, $\{v_1, \ldots, v_N\}$. By the incidence matrix $I(\Gamma)$, we mean the matrix $[a_{ij}]$, where $a_{ij} = 1$ if v_iv_j is an edge of Γ, or $i = j$, and $a_{ij} = 0$ otherwise. One useful property of $I(\Gamma)$ is

The (i,j)-th entry of $I(\Gamma)^k$ is the number of paths of length $\leq k$ from v_i to v_j in Γ.

Consequently, we have that the diameter of Γ is the least value of k for which all the entries of $I(\Gamma)^k$ are non-zero. It is this criterion which is used in the algorithm to compute the diameter, since matrix powers are readily computable by a machine.

It turns out that the diameter alone is itself not a sufficiently sensitive invariant for purposes of the algorithm. Specifically, there are too many graphs for which no perturbation results in an improvement of the diameter. Consequently, using only the diameter as a functional to be optimized, the algorithm is frequently unable to find a graph with even reasonable diameter. To see how to improve matters, we need a definition. Let $\mathbb{Z}$ denote the integers. We wish to define an ordering on $\mathbb{Z}^n$, called the lexicographic ordering. For $n = 1$, the lexicographic ordering on $\mathbb{Z}^n = \mathbb{Z}$ is just the usual ordering on the integers. For $n > 1$, we suppose the ordering is already defined for all $m < n$. We write $\mathbb{Z}^n = \mathbb{Z} \times \mathbb{Z}^{n-1}$, and define the ordering inductively by

$$(z,w) < (z',w') \Leftrightarrow \begin{cases} z < z' & \text{or} \\ z = z' & \text{and} \quad w < w' . \end{cases}$$

for $z \in \mathbb{Z}$, $w \in \mathbb{Z}^{n-1}$. Now, we will associate to every graph its "diameter vector." First, for a positive integer ℓ, we define $\alpha_\ell(\Gamma)$ to be the number of zeros in the ℓ-th power of $I(\Gamma)$. Of course, if $\ell > \Delta(\Gamma)$, $\alpha_\ell(\Gamma) = 0$. The diameter vector is now simply the vector ~~is now simply the vector~~

$$\Big(\Delta(\Gamma)(=k),\ \alpha_{k-1}(\Gamma),\ \alpha_{k-2}(\Gamma),\ \ldots,\ \alpha_2(\Gamma)\Big) .$$

We will denote this by $v(\Gamma)$. We order these vectors as follows:

$$\Big(\Delta(\Gamma),\ \alpha_{k-1}(\Gamma),\ \ldots,\ \alpha_2(\Gamma)\Big) \le \Big(\Delta(\Gamma'),\ \alpha_{k'-1}(\Gamma),\ \ldots,\ \alpha_2(\Gamma)\Big)$$

$$\Leftrightarrow \begin{cases} \Delta(\Gamma) < \Delta(\Gamma') & \text{or} \\ \Delta(\Gamma) = \Delta(\Gamma') & \text{and} \quad \alpha(\Gamma) \le \alpha(\Gamma') \end{cases}$$

Here, $\alpha(\Gamma)$ denotes the vector $\left(\alpha_{k-1}(\Gamma), \ldots, \alpha_2(\Gamma) \right)$, and the ordering is the lexicographic ordering.

The algorithm now proceeds as follows. We say a 4-tuple (v_1, v_2, w_1, w_2) is Γ-admissible if v_1v_2 and w_1w_2 are edges of Γ, and v_1w_1 and v_2w_2 are not. To a Γ-admissible 4-tuple we may associate a perturbation of Γ, as defined above. From a fixed initial graph Γ, Γ-admissible 4-tuples are generated, and the associated perturbations are applied. This continues for a large number of steps, until the initial graph is presumed randomized. The 4-tuples are generated using a random number generator. The fixed initial graph (in the trivalent case) is an n-cycle with antipodal points connected. After this is done, the steps are as follows:

(I) Select a Γ-admissible 4-tuple at random.

(ii) Compute $V(\tilde{\Gamma})$, where $\tilde{\Gamma}$ is the graph obtained by applying the perturbation associated to the 4-tuple constructed in (I). If $v(\tilde{\Gamma}) < v(\Gamma)$, set $\Gamma = \tilde{\Gamma}$. Repeat step (I).

The perturbations are selected at random, since it was found that a simple ordering of perturbations tended to bias the algorithm toward particular graphs.

We compare our algorithm to that devised in [T-S]. Our perturbations are precisely their X-changes, but the functional we optimize is much more sensitive. Theirs consists only of $\Delta(\Gamma)$ and $\alpha_{k-1}(\Gamma)$.

We now summarize the results of the application of our algorithm.

By the use of the algorithm, it has been possible to improve substantially most of the densest known graphs. We give our improved version of the table constructed in [LFQSU]. d denotes the degree of the graph, k the diameter.

The (d,k) entry is the largest known graph with diameter k and degree d. Our entry is listed above, the parenthesized value below is the value from [LFSQU]. One asterisk indicates that the graph is probably optimal. Two asterisks indicates that it is obtained from [B] using the result cited in §I.

The results obtained from this algorithm are in some cases surprising. We list some of the qualitative properties observed.

(a) Many graphs obtained by <u>random</u> generation of graphs improved values in the older version of the table in [LFQSU], given by Storwick [S]. This suggests that one is further from optimal than was previously thought.

(b) By evaluating the eigenvalues of the incidence matrices arrived at by the algorithm, it was found that there are many distinct "local minima" (i.e., graphs for which no perturbation improves the diameter vector) for the diameter vector. This contradicts the suggestion made in [T-S], that one tends to arrive at a global optimum from all starting points. It seems that the algorithm in [T-S] suffers from two deficiencies. First, their objective functional for minimization is not sufficiently sensitive, as we observed above. Secondly, their perturbations are done in fixed sequential order, which severely skews their results. We have overcome this difficulty by randomly selecting the perturbations at each stage.

(c) Although our algorithm is efficient, it seems that substantially larger networks could be studied if our diameter routine were modified to use the so-called "Dijkstra algorithm," which would speed up the diameter calculation substantially.

d \ k	2	3	4	5	6	7	8	9	10
3	10* [10]	20* [20]	38 [34]	58 [56]	125** [84]	140 [122]	240 [176]	311 [311]	609 [525]
4	15* [15]	36 [35]	80 [67]	134 [134]	728** [261]	728** [425]	910 [910]	1,520 [1,360]	2,888 [2,312]
5	24* [24]	50 [48]	170* [126]	262 [262]	2,730** [505]	2,730** [1,260]	2,730** [2,450]	5,760 [4,690]	9,648 [9,380]
6	31 [31]	80 [65]	312** [164]	600 [600]	7,812** [1,152]	7,812** [2,520]	7,812** [6,561]	19,683 [19,683]	59,049 [59,049]
7	50* [50]	88 [88]	312** [252]	992 [992]	7,812** [2,880]	7,812** [4,680]	12,960 [12,250]	43,200 [43,200]	90,000 [86,400]
8	57 [57]	106** [105]	800** [384]	2,550 [2,550]	39,216** [5,760]	39,216** [16,384]	65,536 [65,536]	262,144 [262,144]	1,048,576 [1,048,576]
9	74 [74]	150 [150]	1,160** [600]	3,306 [3,306]	74,898** [12,500]	74,898** [20,160]	76,500 [76,500]	382,500 [382,500]	1,048,576 [1,048,576]
10	91 [91]	200 [200]	1,640** [864]	5,550 [5,550]	132,860** [25,000]	132,860** [78,125]	390,625 [390,625]	1,953,125 [1,953,125]	9,765,625 [9,765,625]

Figure 1

(d) Although the algorithm is an improvement over all previous heuristic algorithms for this problem, it is unable to find many known dense graphs, arising from systematic constructions. The reason for this seems to be the "denseness" of the set of local optima in the set of all graphs of degree d, and the relative sparseness of the so-called vertex transitive graphs therein. It seems, therefore, that it would be desirable to design an algorithm which operates entirely inside a collection of vertex transitive graphs, possibly within the Cayley graphs (see §D). Using the Dijkstra diameter algorithm, and an efficient description of many groups, such an algorithm should be constructible. Moreover, it would allow much larger networks to be studied, since the diameter calculation for vertex transitive graphs is substantially shorter than that for arbitrary graphs.

A copy of the code for the implementation of this algorithm is enclosed, as well as documentation for it.

(C) Modification of the Algorithm for More Sensitive Measures

In view of the remarks in §I which identify the measures $\Delta_k^{\ell}(\Gamma)$ as the diameter of an associated graph Γ_k^{ℓ}, one can in principle study these measures using the algorithm discussed in §B. However, the graphs Γ_k^{ℓ} are usually too large for this procedure to be practicable. Our current implementation of the algorithm will accept only graphs with fewer than 10,000 points, and Γ_k^{ℓ} usually is larger than this. For the measures Δ_k^{ℓ}, we therefore propose the use of a "dual algorithm" based on girth, which is much simpler to compute (see §A).

The modified girth algorithm is identical to the previous algorithm, except that the objective functional is altered. For a graph Γ, we define its girth vector to be $\gamma(\Gamma) = \left(\nu_1(\Gamma), \nu_2(\Gamma), \ldots, \nu_k(\Gamma), \ldots \right)$. This is ordered by the lexicographic ordering, and the algorithm proceeds just as before, except that we now accept a perturbation if it increases $\gamma(\Gamma)$. Applying this algorithm to Γ_k^{ℓ} should produce heuristic results which improve these measures.

(D) Vertex-Transitive Graphs

One desirable feature of a graph to be used as a processor array is that the description be as simple and compact as possible. Thus, the graphs produced by a heuristic algorithm will in general not be satisfactory from this point of view, and for this purpose it would be useful to restrict oneself to a class of graphs having a compact description.

One such family is the collection of so-called Cayley graphs. Let us define these. Let G be a finite group, and let $\Omega \subseteq G$ be a subset, closed under inversion, so that $\Omega = \Omega^{-1}$. Then we define the Cayley graph associated to the pair (G,Ω), $\Gamma(G,\Omega)$, to be the graph whose vertices are the elements of G, and whose edges are pairs $(e,\omega g)$ for $\omega \in \Omega$. As an example, if $G = \mathbb{Z}/n$, the cyclic group with n elements, and $\Omega = \{T,T^{-1}\}$, where T is a generator of G, the associated graph is the cyclic graph of size n. A useful property of $\Gamma(G,\Omega)$ is that its automorphism group is transitive on the vertices, i.e., for every pair (x,y) of vertices of $\Gamma(G,\Omega)$, there is an automorphism α of $\Gamma(G,\Omega)$ so that $\alpha(x) = y$. This is clearly the case since the right G-action on G provides an action of G on the graph, which is clearly transitive on the vertex set. This is a useful property, since it means that the diameter may be computed by finding the points of maximal distance from one given point. Also, routing algorithms for these networks are compactly described, since one must only find optimal paths starting at one given point.

One important proposed architecture, the cube-connected cycles of [P-V], are in fact of this form. For, consider the semi-direct product $G = \mathbb{Z}/n \underset{\rho}{\times} \mathbb{Z}/2^n$, $\rho(T)(x_1, \ldots, x_n) = (x_n, x_1, \ldots x_{n-1})$. Thus G has

elements (m,v), where $m \in \mathbb{Z}/n$, and $v \in (\mathbb{Z}/2)^n$, and $(m,v)(m',v') = (m+m', \rho(m')v + v')$. It is an easy calculation to see that if $\Omega = \{(1,0), (-1,0), (0,e_1)\}$, where $e_1 = (1,0,0,\ldots,0)$, then $\Gamma(G,\Omega)$ is in fact isomorphic to the graph associated with the cube-connected cycles. The diameter of the cube-connected cycles is known to grow as $5/2 \log_2(K)$, where K is the number of vertices in the graph.

We should remark here that large girth (and hence small diameter) in Cayley graphs is associated with non-commutativity of the group in question. This being the case, the simple groups seem to be natural candidates to produce efficient graphs. To see that this is in fact the case, we study the diameters of $\Gamma(G,\Omega)$ for certain choices of Ω, and $G = S_n$, and observe that for large n, they approximate the Moore bound. The order of the symmetric group S_n is $n!$. Let $\Omega_k \subseteq S_n$ be the set of all cycles of length $\leq k$. We propose to compare the diameter of $\Gamma(S_n,\Omega_k)$ with its associated Moore bound. First we observe that $|\Omega_k| = \binom{n}{2} + 2\binom{n}{3} + 6\binom{n}{4} + \ldots + (k-1)!\binom{n}{k}$, by a simple counting argument. Thus, the degree of $\Gamma(S_n,\Omega_k)$ is $d = \sum_{j=2}^{k} (j-1)!\binom{n}{j}$. Thus for large n, the Moore bound for $\Gamma(S_n,\Omega_k)$ is $\log_{d-1}(n!) = \frac{\ell n(n!)}{\ell n(d-1)}$. By Stirling's formula, $\lim_{n\to\infty} \frac{\ell n(n!)}{n \ell n\, n} = 1$, so for large n, the Moore bound is approximated by $\frac{n \ell n\, n}{\ell n(d-1)}$. But, again for large n, $\ell n(d-1) = \ell n\left(\sum_{j=2}^{k} (j-1)!\binom{n}{j}\right)$ is approximated by $\ell n\binom{n}{k} \approx k \ell n\, n$.

Hence, for large n, the Moore bound for $\Gamma(S_n,\Omega_k)$ is approximated by n/k.

The diameter of (S_n,Ω_k), on the other hand, tends to $n/k-1$, as one readily computes in S_n. Consequently, the diameter of (S_n,Ω_k) is within a factor of $k/k-1 = 1 + \frac{1}{k-1}$ of the Moore bound. As k becomes larger, we are able to approximate equality arbitrarily closely. This shows that S_n is a plausible candidate for further study.

We will now show how to modify the cube-connected cycles, using group theoretic methods, to provide an infinite family of vertex transitive trivalent graphs, whose diameter is substantially smaller, but which have all the desirable regularity properties of the cube-connected cycles. Let $G_n = \mathbb{Z}/n \underset{\rho}{\times} (\mathbb{Z}/2)^n$, as before. Note that G_n contains a central element, namely the vector $(0,(1,1,\ldots,1))$. Following our intuition concerning the relationship between non-commutativity and small diameter, we eliminate the central element by simply factoring it out. Call the quotient group $\tilde{G}_n$, and let $\tilde{\Omega}$ denote the image of Ω. Then we claim that the diameter of $\Gamma(\tilde{G}_n,\tilde{\Omega})$ grows as $2\log_2(K)$, where K is the number of vertices, an improvement over the diameter $5/2\log_2(K)$ obtained for the cube-connected cycles.

Proposition The diameter of $\Gamma(\tilde{G}_n,\tilde{\Omega})$ grows as $2\log_2(K)$

Proof By taking inverses, we can clearly consider the graph $\overline{\Gamma}(\tilde{G}_n,\tilde{\Omega})$, where $(g,g\omega)$ is an edge for $\omega \in \Omega$. An element of $\tilde{G}_n$ is given by an ordered pair (m,v), $m \in \mathbb{Z}/n$, $v \in \bar{V} = (\mathbb{Z}/2)^n/(1,\ldots,1)$. The multiplication is given by $(m,v)e_i = (m, v+e_i)$, $(m,v)T = (m+1, \rho(T)v)$, $(m,v)T^{-1} = (m-1, \rho(T^{-1})v)$. An algorithm for expressing (m,v) in terms of the generators T, T^{-1}, and e_i is given as follows. Let $x_i(v)$ denote

the first coordinate of a vector $v \in V = (\mathbb{Z}/2)^n$. For any $v \in \overline{V}$, one may lift v to an element $\tilde{v}$ of V, so that the number of non-zero coordinates in $\tilde{v}$ is $\leq n/2$, for if one lift $\tilde{v}$ does not have this property, then $\tilde{v} + (1,1,\ldots,1)$ does. Given (m,v), select $\tilde{v}$ as above.

The algorithm now proceeds as follows.

(I) Initialize a counter α at $n-1$.

(II) Is $x_1(\tilde{v}) = 0$? If yes, proceed to (IV), if no, proceed to (III).

(III) Multiply by e_1. Proceed to (IV).

(IV) Multiply by T. $\alpha - 1 \to \alpha$. Proceed to (V).

(V) Is $\alpha = 0$? If so, proceed to (VI). If not, return to (II).

(VI) If $m \neq 0$, multiply by T^q, where q is the number of minimal absolute value congruent to $-m \mod n$. Note that $|q| \leq n/2$. Quit.

Since $\tilde{v}$ has at most $n/2$ 1's, we only multiply by e_1 at most $n/2$ times. Thus, the total number of steps is at most $(n-1) + n/2 + n/2 = 2n - 1$. For odd n, it is at most $2n - 3$, which is of the same order as $2 \log_2(n\, 2^{n-1})$.

□

If one forms the quotient of this graph by the equivalence relation $(m,v) \approx (m',v)$, $\forall\, m,m' \in \mathbb{Z}/n$, one obtains a non-regular family of graphs whose diameter grows as $3/2 \log_2(n)$, which is comparable to that obtained in [L-S], and for which the routing algorithm is much simpler. Finally, an alternative version of this construction is given by forming the $(n-1)$-cube, inserting n-cycles at every vertex so that the incoming edges each connect at distinct vertices, and connect the remaining vertex to the

corresponding vertex for the antipodal point on the cube. This is illustrated below for $n = 3$

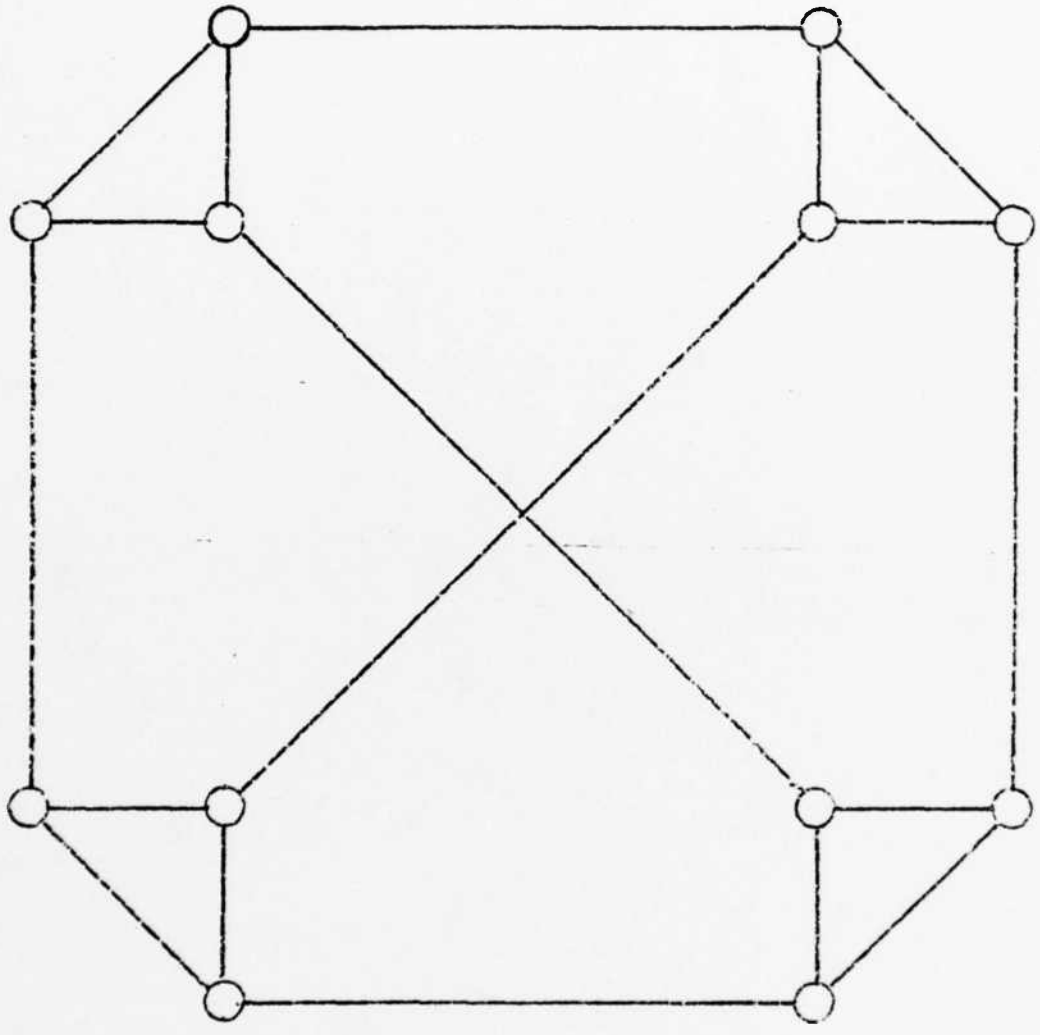


Figure 2

(E) A Layout for the Modified Cube-Connected Cycles

In the paper [P-V], two layouts are proposed for the cube connected cycles, one slightly more efficient than the other. By combining these two layouts, we obtain a layout for the modified cube connected cycles. The area of the layout grows as 3/2 times the area of the cube connected cycles with the same number of nodes, and has communication time roughly 4/5 times that of the cube connected cycles. We give the layout for the case $n = 5$, corresponding to $5 \cdot 2^4 = 80$ nodes. It will be clear from the diagram how to extend to the general case.

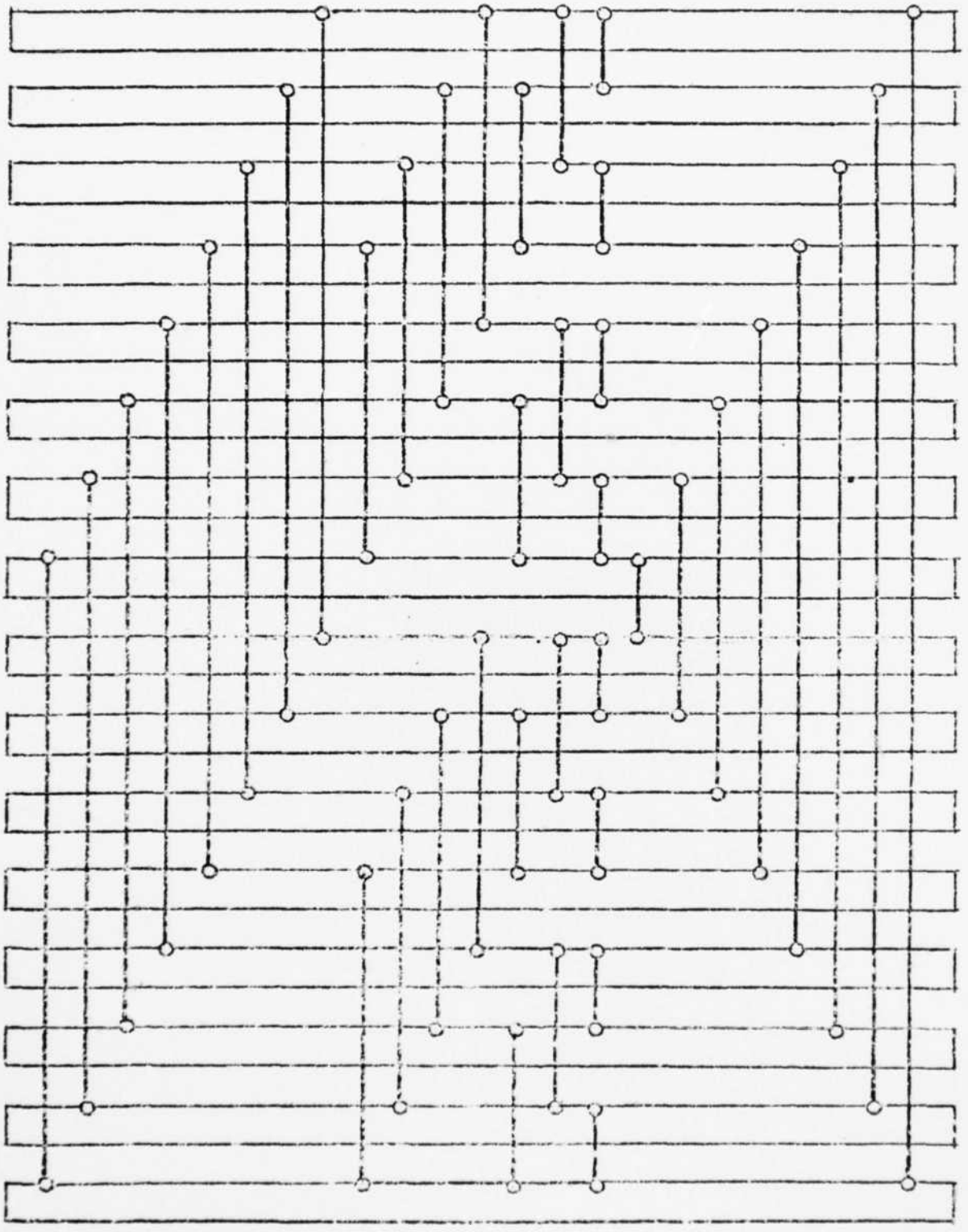


Figure 3

REFERENCES

[B] Biggs, N., Algebraic Graph Theory, Cambridge University Press, 1974.

[Bo] Bollobás, B., Extremal Graph Theory, Academic Press, 1978.

[LFQSU] Leland, W., Finkel, R., Qiao, L., Solomon, M., and Uhr, L. "High density graphs for processor interconnection," Information Processing Letters, Vol. 12, No. 3 (1981).

[L-S] Leland, W. and Solomon, M. "Dense trivalent graphs for processor interconnection," IEEE Transactions on Computers, Vol. C-31, No. 3, March, 1982.

[P-V] Preparata, F. and Vuilliemin, J., "The cube-connected cycles: a versatile network for parallel computation," Communications of the ACM, Vol. 24, No. 5., May, 1981.

[S] Storwick, R. M., "Improved construction techniques for (d,k) graphs," IEEE Transactions on Computers, 19 (12) (1970).

[T-S] Toneg, S. and Steiglitz, K., "The design of small-diameter networks by local search," IEEE Transactions on Computers 28 (7) (1979).

```
CFLAGS = -O

search  :       search.o zeroes.o eigen.o geneval.o
                cc $(CFLAGS) search.o zeroes.o eigen.o geneval.o -lm -o search

search.o :      declar.h search.c
                cc $(CFLAGS) -c search.c

zeroes.o :      declar.h zeroes.c
                cc $(CFLAGS) -c zeroes.c

eigen.o :       declar.h eigen.c
                cc $(CFLAGS) -c eigen.c

geneval.o:      declar.h geneval.c
                cc $(CFLAGS) -c geneval.c
```

```
#define MOD %
#define Intsize (8 * sizeof(long))
#define Maxsize 256
#define Maxreduced Maxsize/(Intsize)
#define YES 1
#define NO 0
#define SCREENSZ  80 /* the width of the screen; for I/O control */
#define EVSIZE  6 /* the width of the field of an e.v. display */
#define DISPCT (SCREENSZ / EVSIZE)

extern int (*multiply)();
extern int multgraph();
extern int mult1graph();
extern int mult2graph();

extern int (*nextgraph)();
extern int se[illegible]ntial();
extern int rn[illegible]_move();

extern int (*evaluategraph)();
extern int eval_diamvect();
extern int eval_passall();

typedef long Graph[Maxsize][Maxreduced];
typedef double Matrix[Maxsize][Maxsize];
typedef double Vector[Maxsize];
typedef long Passgraph[][Maxreduced];

extern int numpts;          /* the number of pts in the graph */
extern int degree;
extern long place[Maxsize];
extern long graphA[Maxsize][Maxreduced];
extern long graph[2][Maxsize][Maxreduced];
extern double matrix[Maxsize][Maxsize];
extern double evvec[Maxsize];
extern int vectorA[Maxsize], vectorB[Maxsize];
extern int *vectA, *vectB, *temp;
extern int diamA, diamB;
extern int x1, x2, x3, x4;          /* coords. for the current perturbation */
extern struct Status {
  int starting;          /* =YES => start up procedures for the program */
  int continue;          /* =NO => don't look for any more graphs */
  int newgraph;          /* =YES => last perturbation was accepted */
  long graphcount;       /* # of new graphs produced */
  int printing;          /* =NO => no printing of intermediate results */
  int diamwatch;         /* to watch diam. results for unsuccessful
                    comparisons set diamwatch = YES  */
  int backtracking;      /* =YES => backtracking in effect */
  int eigenvalues;       /* =YES => find all eigenvalues */
  } status;
extern double Sqrt2inv;
extern long nextprint, increment, incincrement, incfactor;
extern long lowerlimit, upperlimit;
```

```
#include <math.h>

#include "declar.h"

double matrix[Maxsize][Maxsize];
double evvec[Maxsize];

compar(num1, num2)                                              compar
double *num1, *num2;
/* this procedure compares two real numbers, returning -1 if num1 >
num2, 0 if num1=num2, and +1 if num1 < num2. Big deal. (It is needed
for qsort in cyclic_jacobi.) */
{
  if (*num1 > *num2)
    return(-1);
  else if (*num1 == *num2)
    return(0);
  else
    return(1);
}

/*************************************************************************/

cyclic_jacobi(graph1)                                           cyclic_jacobi
Passgraph graph1;
/* this procedure takes graph1, which is the "augmented" incidence matrix
of a graph, strips out the diagonal 1's, leaving the usual incidence matrix,
transforms the "bit level" matrix to one with real entries (only the upper
half of the matrix is used anywhere), and then uses
the cyclic Jacobi method to find the eigenvalues. It also computes some
measure of the degeneracy (still to be determined at this printing).
"evvec1" is the quicksorted diagonal of the matrix. It
seems very likely that for large graphs better methods using the sparsity
of the incidence matrix would be valuable. */
{
#define EPSILON 0.000001
#define ITERBD 100
  double error;
  int itnum;
  double *ai;
  long *gi;
  int i, j;

  for(i=0; i < numpts; i++)
  {
    ai = matrix[i];
    *(ai + i) = 0.0;
    gi = graph1[i];
    for(j=i+1; j < numpts; j++)
      if (*(gi + (j/Intsize)) & place[j])
        *(ai + j) = 1.0;
      else
        *(ai + j) = 0.0;
  }
  error = 1.0;
  for (itnum = 0; error > EPSILON; itnum++)
  {
    cjiteration(matrix, &error);
    if (itnum > ITERBD)
    {
      printf("CJ Alg. looping too many times.\n");
      return;
    }
```

```
  }
  for(i=0; i < numpts; i++)
    evvec[i] = matrix[i][i];

  qsort(evvec, numpts, sizeof(double), compar);

  evprint(evvec);
}

/*****************************************************************/

cjiteration(matrix1,errpt)                                    cjiteration
Matrix matrix1;
double *errpt;
/*this procedure performs one iteration of the cyclic Jacobi algorithm
on the matrix matrix2. The error is the sum of the squares of the
off-diagonal elements. Notice that only the upper half of the matrix is
ever used.*/
{
#define DELTA 0.000000000001
  int i, j, k;
  double q, v;
  double alpha, beta;
  double alphasqr;
  double aij, aijsqr;
  double aii, ajj;
  double dmi, dmj;
  double temp;
  double *ai, *aj;

  for(i=0; i < numpts; i++)
  {
    ai = matrix1[i];
    for(j=i+1; j < numpts; j++)
    {
      aij = *(ai + j);
      aijsqr = pow(aij, 2.0);
      if (aijsqr > DELTA)
      {
        aj = matrix1[j];
        aii = *(ai + i);
        aij = *(aj + j);
        q = aii - ajj;
        v = sqrt( (4.0 * aijsqr) + pow(q, 2.0) );
        if (q == 0.0)
        {
          beta = alpha = Sqrt2inv;
          alphasqr = 0.5;
        }
        else
        {
          alphasqr = fabs(v + q) / (2.0 * v);
          alpha = sqrt(alphasqr);
          beta = aij / (v * alpha);
        }
        for(k=0; k < i; k++)
        {
          dmi = matrix1[k][i];
          dmj = matrix1[k][j];
          matrix1[k][i] = alpha * dmi + beta * dmj;
          matrix1[k][j] = alpha * dmj - beta * dmi;
        }
        temp = (q * alphasqr) + (2.0 * alpha * beta * aij);
        *(ai + i) = ajj + temp;
```

...cjiteration

```
        for(k=i+1; k < j; k++)
        {
          dmi = *(ai + k);
          dmj = matrix1[k][j];
          *(ai + k) = alpha * dmi + beta * dmj;
          matrix1[k][j] = alpha * dmj - beta * dmi;
        }
        *(ai + j) = 0.0;
        *(aj + j) = aii - temp;
        for(k=j+1; k < numpts; k++)
        {
          dmi = *(ai + k);
          dmj = *(aj + k);
          *(ai + k) = alpha * dmi + beta * dmj;
          *(aj + k) = alpha * dmj - beta * dmi;
        }
      }
    }
  }

  *errpt = 0.0;
  for(i=0; i < numpts; i++)
    for(j=i+1; j < numpts; j++)
      *errpt += pow(matrix1[i][j], 2.0);
}

/*****************************************************************/
```

evprint

```
evprint(evvec2)
Vector evvec2;
/* this procedure prints the eigenvalue information obtained by
cyclic_jacobi, above. */
{
  int i;

  printf("Number of nodes:%1d, Degree: %1d,  ",numpts, degree);
  printf("Eigenvalues of the incidence matrix:\n");
  for(i=0; i<numpts; i++)
    printf("%7.3f%s",evvec2[i],((i MOD 10 == 9)||(i==numpts - 1)) ? "\n" : "");
}
```

```
#include "declar.h"

generategraph()                                              generategraph
{

  if (status.starting == YES)
  {
    startgraph(graphA, &numpts);
    return;
  }
  (*nextgraph)(graphA);
}

/*****************************************************************/

sequential(graph1)                                           sequential
Passgraph graph1;
/* takes graph1 and identifies the next sequential perturbation,
   performs it, and sets x1, x2, x3, and x4 to identify it */
{
  static int i;
  static int y3, y4;
  static int z1;
  static int firsttime;
#define Backlimit 30
  static int a1[Backlimit], a2[Backlimit], a3[Backlimit], a4[Backlimit];
  static int c3[Backlimit], c4[Backlimit], j[Backlimit], b1[Backlimit];
  static int level = 0;

  if (status.newgraph == YES)
  {
    firsttime = YES;
    if (status.graphcount != 1L)
    {
      a1[level] = x1;
      a2[level] = x2;
      a3[level] = x3;
      a4[level] = x4;
      c3[level] = y3;
      c4[level] = y4;
      j[level] = i;
      b1[level] = z1;
      level = (level + 1) MOD Backlimit;
      goto resume1;
    }
  }
  else
    goto resume;
  level = 0;
  for ( ; ; )
  {
    for (x1=1; x1 < numpts - 1; x1++)
    {
      if (firsttime == YES)
      {
        firsttime = NO;
        z1 = x1;
      }
      else if (z1 == x1)
        if (status.newtracking == YES)
        {
          printf("Your (possibly) favorite graph\n");
```

```
            printgraph(graph1,:numpts);
            cyclic_jacobi(graph1);
            if (diamA == 4)
              takepo(graph1, vectA, numpts, &diamA, numpts);
            status.graphcount--;
            nextprint--;
            increment = 1L;
            incfactor = 1L;
            incincrement = 1L;
            lowerlimit = 0L;
            upperlimit = 1000000L;
            diaminfo(vectA, diamA);
            level = (level - 1 + Backlimit)%MOD%Backlimit;
            x1 = a1[level];
            x2 = a2[level];
            x3 = a3[level];
            x4 = a4[level];
            y3 = c3[level];
            y4 = c4[level];
            i  = j[level];
            z1 = b1[level];
            perturb(graph1, x1, x4, x3, x2);
            takepowers(graph1, vectA, numpts, &diamA, numpts);
            for (i=0; i < Maxsize; i++)
              *(vectB + i) = *(vectA + i);
            diamB = diamA;
            goto resume;
          }
          else
          {
            status.contnue = NO;
            return;
          }
        for (x2=0; x2 < x1; x2++)
          if (graph1[x1][x2/(Intsize)] & place[x2])
            for (y3=x1 + 1; y3 < numpts; y3++)
              for (y4=0; y4 < y3; y4++)
                if ((graph1[y3][y4/(Intsize)] &place[y4])
                  && (x1 != y4) && (x2 != y4))
                  for (i=0; i <= 1; i++)
                  {
                    if (i==0)
                    {
                      x3 = y3;
                      x4 = y4;
                    }
                    else
                    {
                      x3 = y4;
                      x4 = y3;
                    }
                    if ( !(graph1[x2][x3/(Intsize)] & place[x3])
                      && !(graph1[x1][x4/(Intsize)] & place[x4]))
                    {
                      perturb(graph1, x1, x2, x3, x4);
                      return;
                    }
resume:
                    ;
                  }
resum1:
      ;
    }
  }
```

...sequential

```
}

/*****************************************************************/

rndm_move(graph1)                                        rndm_move
Passgraph graph1;
/* randomly picks the next perturbation, performs it, and records
   it using x1, x2, x3, x4 */
{
  int i, j;

  for ( ; ; )
  {
    x1 = rand() MOD numpts;
    j  = rand() MOD degree;
    for (i=0; i < numpts; i++)
      if ( (i != x1) && (graph1[x1][i/Intsize] & place[i]) )
        if (j==0)
        {
          x2 = i;
          break;
        }
        else
          j--;
    for (x3=rand() MOD numpts; (x3 == x1) || (x3 == x2); x3=rand() MOD numpts);
    j = rand() MOD degree;
    for (i=0; i < numpts; i++)
      if ( (i != x3) && (graph1[x3][i/Intsize] & place[i]) )
        if (j == 0)
        {
          x4 = i;
          break;
        }
        else
          j--;
    if ( (x4 != x1) && (x4 != x2) && (!(graph1[x2][x3/Intsize] & place[x3]))
      && (!(graph1[x1][x4/Intsize] & place[x4])) )
    {
      perturb(graph1, x1, x2, x3, x4);
      return;
    }
    else
      continue;
  }
}

/*****************************************************************/

eval_diamvect()                                          eval_diamvect
{
  int i;
  int improved;
  static int diamcompare;

  if (status.starting == YES)
  {
    if (numpts <= Intsize)
      multiply = mult1graph;
    else if (numpts <= 2 * Intsize)
      multiply = mult2graph;
    else
      multiply = multigraph;
    zerocount(graphA, numpts, &i);
    vectorA[1] = i;
```

...eval—diamvect

```
    diamcompare = numpts;
    takepowers(graphA, vectA, numpts, &diamA, diamcompare);
    *(vectB + 1) = *(vectA + 1);
    diamB = diamA;
  }
  else
  {
    if (status.diamwatch == NO)
      diamcompare = diamA;
    takepowers(graphA, vectB, numpts, &diamB, diamcompare);
    compare(vectA, vectB, diamA, diamB, &improved);
    if (improved)
    {
      status.newgraph = YES;
      temp = vectA;
      vectA = vectB;
      vectB = temp;
      diamA = diamB;
      if (status.eigenvalues == YES)
        cyclic_jacobi(graphA);
    }
    else
    {
      perturb(graphA, x1, x4, x3, x2);
      status.newgraph = NO;
    }
  }
  if (status.newgraph == YES)
  {
    status.graphcount++;
    printcontrol();
  }
}

/*****************************************************************/
```

eval—passall

```
eval_passall()
/* this considers all graphs to be improvements */
{
  status.newgraph = YES;
  takepowers(graphA, vectB, numpts, &diamB, numpts);
  temp = vectA;
  vectA = vectB;
  vectB = temp;
  diamA = diamB;
  if (status.eigenvalues == YES)
    cyclic_jacobi(graphA);
  status.newgraph = YES;
  status.graphcount++;
  printcontrol();
}

/*****************************************************************/
```

perturb

```
perturb(graph1, xx1, xx2, xx3, xx4)
Passgraph graph1;
int xx1, xx2, xx3, xx4;
/* this performs the perturbation
   (xx1, xx2), (xx3, xx4) -> (xx2, xx3), (xx1, xx4)  */
{
  graph1[xx1][xx2/(Intsize)] &= ~(place[xx2]);
  graph1[xx2][xx1/(Intsize)] &= ~(place[xx1]);
  graph1[xx3][xx4/(Intsize)] &= ~(place[xx4]);
  graph1[xx4][xx3/(Intsize)] &= ~(place[xx3]);
```

...perturb

```
  graph1[xx1][xx4/(Intsize)] |= place[xx4];
  graph1[xx4][xx1/(Intsize)] |= place[xx1];
  graph1[xx2][xx3/(Intsize)] |= place[xx3];
  graph1[xx3][xx2/(Intsize)] |= place[xx2];
}

/*****************************************************************/
```

compare

```
compare(dumvec1, dumvec2, dumdiam1, dumdiam2, dumimp)
/* This procedure compares (starting at the bottom) dummy vec1 with
dummy vec2 and if 2 is better, dumimp is set to 1. */
int *dumimp;
int dumvec1[], dumvec2[];
int dumdiam1, dumdiam2;
{
  int i ;

  if (dumdiam2 < dumdiam1)
  {
    *dumimp = 1;
    return;
  }
  *dumimp = 0;
  for (i=dumdiam1; i >= 2; i--)
  {
    if (dumvec2[i] < dumvec1[i])
    {
      *dumimp = 1;
      return;
    }
    else if (dumvec1[i] < dumvec2[i])
    {
      *dumimp = 0;
      return;
    }
  }
}
```

```
#include <math.h>

#include "declar.h"

int (*multiply)();
int multgraph();
int mult1graph();
int mult2graph();

int (*nextgraph)();
int sequential();
int rndm_move();

int (*evaluategraph)();
int eval_diamvect();
int eval_passall();

int numpts;        /* the number of pts in the graph */
int degree;
long place[Maxsize];
long graph[2][Maxsize][Maxreduced];
long graphA[Maxsize][Maxreduced];
int vectorA[Maxsize], vectorB[Maxsize];
int *vectA, *vectB, *temp;
int diamA, diamB;
int x1, x2, x3, x4;          /* coords. for the current perturbation */
struct Status status = {
  YES, YES, YES, 0L, YES, NO, YES, NO};
/*
  int starting;          /* =YES => start up procedures for the program */
  int continue;          /* =NO => don't look for any more graphs */
  int newgraph;          /* =YES => last perturbation was accepted */
  long graphcount;       /* # of new graphs produced */
  int printing;          /* =NO => no printing of intermediate results */
  int diamwatch;         /* to watch diam. results for unsuccessful
                         comparisons set diamwatch = YES */
  int backtracking;      /* =YES => backtracking in effect */
  int eigenvalues;       /* =YES => find all eigenvalues */
*/
double Sqrt2inv;
long nextprint =1L, increment = 1L, incincrement = 0L, incfactor = 1L;
long lowerlimit = 1L, upperlimit = 1000000L;

main()
{
  int i;

  evaluategraph = eval_diamvect;
  nextgraph = sequential;

  srand(1);

  Sqrt2inv = 1.0 / sqrt(2.0);
  vectA = vectorA;
  vectB = vectorB;

  place[0] = 1L;
  for (i=1; i < Intsize; i++)
    place[i] = 2 * place[i-1];
  for (i=Intsize; i < Maxsize; i++)
    place[i] = place[i MOD (Intsize)];

  for ( ; status.continue == YES; )
```

```
  {
    generategraph();
    if (status.contnue == NO)
      break;
    (*evaluategraph)();
  }
  printf("Our favorite graph:\n");
  printgraph(graphA, numpts);
  printf("\n");
  takepowers(graphA, vectA, numpts, &diamA, numpts);
  diaminfo(vectA, diamA);
}

/*****************************************************************/

printcontrol()                                          printcontrol
{
  int reply, okreply;

  if (status.printing == NO)
    return;
  if (status.graphcount == nextprint)
  {
    if (status.graphcount >= lowerlimit)
      diaminfo(vectA, diamA);
    nextprint += increment;
    increment = incfactor * increment + incincrement;
  }
  if (status.starting == YES || status.graphcount >= upperlimit)
  {
    status.starting = NO;
    for (okreply = NO; okreply == NO; )
    {
      printf("How would you like the intermediate results printed?\n");
      printf("No intermediate results printed (0).\n");
      printf("Doubling of the interval between prints (1).\n");
      printf("A print at every change of diameter (2).\n");
      printf("A semi-custom print interval (3).\n");
      printf("Print at each step(4).\n");
      printf("What is your choice? : ");
      scanf("%d", &reply);
      printf("\n");
      switch (reply)
      {
      case 0:
        status.printing = NO;
        nextprint = 0L;
        return;
      case 1:
        nextgraph = sequential;
        evaluategraph = eval_diamvect;
        nextprint = status.graphcount + 1;
        lowerlimit = 0L;
        upperlimit = 1000000L;
        increment = 1L;
        incincrement = 0L;
        incfactor = 2L;
        okreply = YES;
        break;
      case 2:
        printf("option 2 is not available yet\n");
        break;
      case 3:
        printf("The current graph count is: %ld\n", status.graphcount);
```

```
    printf("Enter the count for the start of printing: ");
    scanf("%ld", &lowerlimit);
    nextprint = lowerlimit;
    printf("Enter the count for reconsidering print control info: ");
    scanf("%ld", &upperlimit);
    printf("After printing starts, printing will occur at\n");
    printf("increments of 'a' to start with, and changed according\n");
    printf("to the folling rule: new = 'b' * old + 'c', ");
    printf("at each step.\n");
    printf("Please enter a, b, c on this line: ");
    scanf("%ld %ld %ld", &increment, &incfactor, &incincrement);
    printf("Which control mechanism would you like for ");
    printf("generating new graphs?\n");
    printf("(1) sequential\n");
    printf("(2) sequential with backtracking\n");
    printf("(3) random\n");
    printf("Your choice:");
    scanf("%d", &reply);
    switch (reply)
    {
    case 1:
      nextgraph = sequential;
      status.backtracking = NO;
      break;
    case 2:
      nextgraph = sequential;
      status.backtracking = YES;
      break;
    case 3:
      nextgraph = rndm_move;
      break;
    default:
      printf("Bad option selected\n");
      break;
    }
    printf("Which evaluation function should be used for accepting");
    printf("new graphs?\n");
    printf("(1) use the zero count vector\n");
    printf("(2) accept all new graphs\n");
    printf("Your choice:");
    scanf("%d", &reply);
    switch (reply)
    {
    case 1:
      evaluategraph = eval_diamvect;
      break;
    case 2:
      evaluategraph = eval_passall;
      break;
    default:
      printf("Bad option selected\n");
      break;
    }
    printf("Indicate whether you are happy (1) or unhappy(2) with ");
    printf("your choices: ");
    scanf("%d", &reply);
    okreply = (reply == 1) ? YES : NO;
    break;
  case 4:
    nextgraph = sequential;
    evaluategraph = eval_diamvect;
    nextprint = status.graphcount + 1;
    lowerlimit = 0L;
    upperlimit = 1000000L;
```

```
        increment = 1L;
        incincrement = 0L;
        incfactor = 1L;
        okreply = YES;
        break;
      default:
        printf("why did you type in %d -- please start over", reply);
        break;
      }
    }
  }
}

/*****************************************************************/
```

printgraph

```
printgraph(graph1, n)
int n;
Passgraph graph1;
/* this procedure prints the graph (incidence matrix) for graph1
   which will be a  n x n  matrix of 0's and 1's */
{
  int i,j;
  char *space;

  printf("\n");
  printf("      ");
  if ( (n + 4) > (SCREENSZ/2) )
    space = "";
  else
  {
    space = " ";
    printf("       ");
  }
  for (i=10; i <= n; i++)
    printf("%s%1d", space, i/10);
  printf("\n");
  for (i=1; i <= n; i++)
    printf("%s%1d", space, i MOD 10);
  printf("\n");
  for (i=0; i < n; i++)
  {
    printf("\n");
    for (j=0; j < n; j++)
      printf("%s%1d", space, (graph1[i][j/(Intsize)] & place[j]) ? 1 : 0);
    printf("  %2d", i+1);
  }
  printf("\n");
}

/*****************************************************************/
```

startgraph

```
startgraph(graph1, n)
Passgraph graph1;
int *n;
/* this procedure generates a starting graph with *n nodes.
   The graph is a ring with opposite sides attached.  */
{
  int i, j;
  int distance;
  int nextpt;

  for ( ; ; )
  {
    printf("Please enter the desired degree of the graph: ");
```

```
    scanf("%d", &degree);
    printf("\n");
    if ((degree < 2) || (degree > Maxsize - 1))
    {
      printf("The degree must be an integer between 2 and %d\n", Maxsize - 1);
      continue;
    }
    printf("please enter the number of nodes you want:");
    scanf("%d", n);
    printf("\n");
    if ((*n <= degree) || (*n > Maxsize))
    {
      printf("The number of nodes must be between %d and %d\n",
      degree + 1, Maxsize);
      continue;
    }
    if ((degree%2 == 1) && (*n%2 == 1))
    {
      printf("For odd degree the number of nodes must be even.\n");
      continue;
    }
    else
      break;
  }

  for (i=0; i<*n; i++)
    for (j=0; j<=((*n)-1)/(Intsize); j++)
      graph1[i][j] = 0L;

  /* this section sets up the ring */
  graph1[0][0] = place[0] ;
  graph1[0][((*n)-1)/(Intsize)] |= place[((*n)-1)];
  graph1[((*n)-1)][0] |= place[0];
  for (i=1; i<*n; i++)
  {
    graph1[i][i/(Intsize)] |= place[i];
    graph1[i][(i-1)/(Intsize)] |= place[(i-1)];
    graph1[i-1][i/(Intsize)] |= place[i];
  }

  /* this section makes the connections across the ring */
  for (i=1; i <= (degree - 1)/2; i++)
  {
    distance = 1 + ((i * ((*n) - 2))/(degree - 1));
    for (j=0; j < *n; j++)
    {
      nextpt = (j + distance)%*n;
      graph1[j][nextpt/(Intsize)] |= place[nextpt];
      graph1[nextpt][j/(Intsize)] |= place[j];
    }
  }
}
```

```
#include "declar.h"

diaminfo(vect1, diam1)                                          diaminfo
int *vect1, diam1;
{
  int i;

  printf("%ld improvements so far.  ", status.graphcount);
  printf("Diameter vector for current min:\n");
  for (i=1; i<=diam1; i++)
    printf("%5d",i);
  printf("\n");
  for (i=1; i<=diam1; i++)
    printf("%5d",*(vect1 + i));
  printf("\n");
}

/*****************************************************************/

takepowers(graph1, vector1, n, diameter, olddiam)               takepowers
Passgraph graph1;
int vector1[];
int n;
int *diameter;
int olddiam;
/* this procedure finds all 'powers' of graph1, which has n pts,
   and forms vector1 which has the count of the number of 0's
   in the corresponding power of graph1.
   The procedure stops taking powers
   when two entries for vector1 are the same, or when the number of
   0's is 0, or when olddiam is seen to be at least as small as the new
   diameter, and returns that as diameter.
*/
{
  int p;
  int numzeroes;
  int evenodd;

  evenodd = 0;
  for (p=1; (p<olddiam)&&(vector1[p] != vector1[p-1])&&(vector1[p]);p++)
  {
    evenodd = 1 - evenodd;
    if (p == 1)
      (*multiply)(graph1, graph1, graph[evenodd], n, &numzeroes);
    else
      (*multiply)(graph1, graph[1-evenodd], graph[evenodd], n, &numzeroes);
    vector1[p + 1] = numzeroes;
  }
  *diameter = p;
}

/*****************************************************************/

takepo(graph1, vector1, n, diameter, olddiam)                   takepo
Passgraph graph1;
int vector1[];
int n;
int *diameter;
int olddiam;
/* this procedure finds all 'powers' of graph1, which has n pts,
   and forms vector1 which has the count of the number of 0's
   in the corresponding power of graph1.
   The procedure stops taking powers
```

```
  when two entries for vector1 are the same, or when the number of
  0's is 0, or when olddiam is seen to be at least as small as the new
  diameter, and returns that as diameter.
*/
{
  int p;
  int numzeroes;
  int evenodd;

  evenodd = 0;
  for (p=1; (p < olddiam) && (vector1[p] != vector1[p -1])&&(vector1[p]); p--)
  {
    evenodd = 1 - evenodd;
    if (p == 1)
      (*multiply)(graph1, graph1, graph[evenodd], n, &numzeroes);
    else
      (*multiply)(graph1, graph[1-evenodd], graph[evenodd], n, &numzeroes);
    printf("\n");
    printgraph(graph[evenodd],.numpts);
    vector1[p - 1] = numzeroes;
  }
  *diameter = p;
}

/*************************************************************************/

zerocount                                                    zerocount(graph1, n, count)
Passgraph graph1;
int n;
int *count;
/* this counts the number of 0's in the matrix for graph1, and returns
   that number in count.  It assumes the matrix is symmetric and has all
   1's on the main diagonal.  */
{
  int i,j;

  *count = 0;
  for (i=0; i<n; i++)
    for (j=i+1; j<n; j++)
      if ( !(graph1[i][j/(Intsize)] & place[j]))
        (*count)++;
  (*count) *= 2;
}

/*************************************************************************/

multgraph                                                    multgraph(graph1, graph2, graph3, n, zerocnt)
Passgraph graph1, graph2, graph3;
int n;
int *zerocnt;
/* this procedure 'multiplies' the matrices for graph1 and graph2, and
   returns the product in graph3.  All matrices are symmetric  n x n. */
{
  int i, j, k;
  int numfields ;
  long *grph3, *grph1;
  long placei;
  int ioverIntsize;

  (*zerocnt) = 0;
  numfields = (n-1)/(Intsize);
  for (i=0; i<n; i++)
  {
    grph1 = *(graph1 + i);
```

...multgraph

```
    grph3 = *(graph3 + i);
    placei = place[i];
    ioverIntsize = i/(Intsize);
    for (j=0; j <= numfields; j++)
      *(grph3 + j) = 0L;
    *(grph3 + ioverIntsize) = placei;
    for (j=0; j < i; j++)
    {
      for (k=0; k <= numfields; k++)
        if ( *(grph1 + k) & graph2[j][k])
        {
          *(grph3 + j/(Intsize)) |= place[j];
          graph3[j][ioverIntsize] |= placei;
          goto nextj;
        }
      (*zerocnt)++;
nextj:
      ;
    }
  }
  (*zerocnt) *= 2;
}

/*****************************************************************/
```

mult1graph

```
mult1graph(graph1, graph2, graph3, n, zerocnt)
Passgraph graph1, graph2, graph3;
int n;
int *zerocnt;
/* this is the same as 'multgraph' except that it's tuned for n<= (intsize) */
{
  int i, j;
  long *grph3, grph1;
  long placei;

  (*zerocnt) = 0;
  *(*(graph3)) = 1L;
  for (i=1; i<n; i++)
  {
    grph1 = *(*(graph1 + i));
    grph3 = *(graph3 + i);
    (*grph3) = placei = place[i];
    for (j=0; j < i; j++)
      if (grph1 & *(graph2[j]))
      {
        *(grph3) |= place[j];
        *(graph3[j]) |= placei;
      }
      else
        (*zerocnt)++;
  }
  (*zerocnt) *= 2;
}

/*****************************************************************/
```

mult2graph

```
mult2graph(graph1, graph2, graph3, n, zerocnt)
Passgraph graph1, graph2, graph3;
int n;
int *zerocnt;
/* this is the same as 'multgraph' except that it's tuned for n > (Intsize) */
```

```
/* and n <= 2 * Intsize */
{
  int i, j;
  long *grph3, *grph1;
  long placei;

  (*zerocnt) = 0;
  grph3 = *graph3;
  (*grph3) = 1L;
  *(grph3 + 1) = 0L;
  for (i=1; i<Intsize; i++)
  {
    grph1 = *(graph1 + i);
    grph3 = *(graph3 + i);
    *(grph3 + 1) = 0L;
    *(grph3) = placei = place[i];
    for (j=0; j < i; j++)
      if ((*grph1 & *(graph2[j])) || (*(grph1 + 1) & graph2[j][1]))
      {
        *(grph3) |= place[j];
        *(graph3[j]) |= placei;
      }
      else
        (*zerocnt)++;
  }
  for (i=Intsize; i < n; i++)
  {
    grph1 = *(graph1 + i);
    grph3 = *(graph3 + i);
    *(grph3) = 0L;
    *(grph3 + 1) = placei = place[i];
    for (j=0; j < i; j++)
      if ((*grph1 & *(graph2[j])) || (*(grph1 + 1) & graph2[j][1]))
      {
        *(grph3 + j/(Intsize)) |= place[j];
        graph3[j][1] |= placei;
      }
      else
        (*zerocnt)++;
  }
  (*zerocnt) *= 2;
}
```

END
DATE
FILMED

MAR 5, 1984